IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences

VOL. E106-A NO. 3
MARCH 2023

A PUBLICATION OF THE ENGINEERING SCIENCES SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER *Special Section on VLSI Design and CAD Algorithms*

Dynamic Verification Framework of Approximate Computing Circuits using Quality-Aware Coverage-Based Grey-Box Fuzzing

Yutaka MASUDA[†a)], ***Member***, **Yusei HONDA**[††], ***Nonmember***, ***and*** **Tohru ISHIHARA**[†], ***Member***

SUMMARY Approximate computing (AC) has recently emerged as a promising approach to the energy-efficient design of digital systems. For realizing the practical AC design, we need to verify whether the designed circuit can operate correctly under various operating conditions. Namely, the verification needs to efficiently find fatal logic errors or timing errors that violate the constraint of computational quality. This work focuses on the verification where the computational results can be observed, the computational quality can be calculated from computational results, and the constraint of computational quality is given and defined as the constraint which is set to the computational quality of designed AC circuit with given workloads. Then, this paper proposes a novel dynamic verification framework of the AC circuit. The key idea of the proposed framework is to incorporate a quality assessment capability into the Coverage-based Grey-box Fuzzing (CGF). CGF is one of the most promising techniques in the research field of software security testing. By repeating (1) mutation of test patterns, (2) execution of the program under test (PUT), and (3) aggregation of coverage information and feedback to the next test pattern generation, CGF can explore the verification space quickly and automatically. On the other hand, CGF originally cannot consider the computational quality by itself. For overcoming this quality unawareness in CGF, the proposed framework additionally embeds the Design Under Verification (DUV) component into the calculation part of computational quality. Thanks to the DUV integration, the proposed framework realizes the quality-aware feedback loop in CGF and thus quickly enhances the verification coverage for test patterns that violate the quality constraint. In this work, we quantitatively compared the verification coverage of the approximate arithmetic circuits between the proposed framework and the random test. In a case study of an approximate multiply-accumulate (MAC) unit, we experimentally confirmed that the proposed framework achieved 3.85 to 10.36 times higher coverage than the random test.

key words: *approximate computing (AC), coverage-based grey-box fuzzing (CGF), design under verification (DUV) integration, verification, computational quality*

1. Introduction

Approximate computing (AC) has recently emerged as a promising approach to the energy-efficient design of digital systems [1]–[7]. While the conventional systems require exact and completely deterministic computation, AC allows some loss of optimality in the computed result. This concept is suitable for a wide range of applications such as digital signal and image processing, wireless communications, and machine learning. By exploiting the inherent resilience of those applications, AC techniques substantially improve energy efficiency (e.g. [1]).

After designing the AC circuit, we need to verify whether the designed circuit can operate correctly under various operating conditions. In this work, we assume the verification where the computational results can be observed, the computational quality can be calculated from computational results with the given metric, and the constraint of computational quality is given and defined as the constraint which is set to the computational quality of designed AC circuit with given workloads. For example, in the image processing accelerator, which receives the input image and outputs the processing image, the above assumptions mean that the output image can be observed, the image quality metric such as Peak Signal-to-Noise Ratio (PSNR) is given, the image quality can be calculated for the output image, and the quality of output image can be compared to the given constraint, e.g., 30dB of PSNR. Under the above assumptions, this paper focuses on the verification of AC circuits whose computational results can be observed and whose computational quality can be calculated from computational results.

Here, the fundamental assumption for the verification is completely different between the conventional circuit and the AC circuit. For example, in the conventional circuit, all paths except for false paths should not cause logic and timing errors. On the other hand, the AC circuit gives a constraint in the computational quality and allows the occurrence of errors as long as the circuit satisfies the target constraint. Namely, in the verification of AC design, we need to find fatal logic errors or timing errors that violate the constraint of computational quality. Originating from the difference of target errors in the verification, the verification technique for the AC circuit is still immature compared with the AC design methodology. Therefore, a novel quality verification methodology for AC circuits is strongly desired.

Quality verification methodologies of the AC circuit can be divided into static and dynamic approaches. As a static approach, formal verification based techniques are proposed in recent years [8]–[10]. These techniques solve the satisfiability (SAT) problems and thus find a correction of errors that violate quality constraints. On the other hand, a dynamic verification technique prepares the test patterns and gives them to the target circuit. After running the test patterns on the circuit, we can obtain the computational results and judge whether the obtained results are acceptable or not. Since the dynamic approach is straightforward [11] and

Manuscript received March 9, 2022.
Manuscript revised July 6, 2022.
Manuscript publicized September 2, 2022.

[†]The authors are with Graduate School of Informatics, Nagoya University, Nagoya-shi, 464-0814 Japan.
[††]The author is with School of Informatics, Nagoya University, Nagoya-shi, 464-0814 Japan.
a) E-mail: masuda@ertl.jp

DOI: 10.1587/transfun.2022VLP0002

easy-to-handle, the dynamic verification through the simulation is one of the most widely deployed approaches [12]. However, in the dynamic approach, how comprehensively the verification is performed heavily depends on the input test pattern. From this point of view, the efficient generation of test patterns is crucially important.

Very recently, for improving the test coverage efficiently, Coverage-based Grey-box Fuzzing (CGF) has been actively developed in the research field of software security testing [13]–[15]. CGF is one of the testing technique called fuzzing [16]–[22]. CGF automatically generates and executes test patterns that enhance verification coverage by repeatedly executing (1) mutation of test patterns, (2) execution of the program under test (PUT), and (3) tabulating code coverage and feeding back to the next test pattern generation. CGF improves the code coverage rapidly by keeping the mutation and coverage aggregation lightweight [16]. This feature is quite attractive not only for software testing but also for hardware verification [23]–[25].

This paper proposes a novel dynamic verification framework of the AC circuit. The key idea of the proposed framework is to incorporate a quality assessment capability into the CGF. We found that CGF originally cannot consider the computational quality by itself. In this case, the mutation could not generate test patterns efficiently that violate the constraint. For overcoming this quality unawareness in CGF, the proposed framework additionally embeds the Design Under Verification (DUV) component into the calculation part of computational quality. Thanks to the integrated DUV component, the proposed framework realizes the quality-aware feedback loop in CGF and thus quickly enhances the verification coverage for test patterns that violate the quality constraint.

Figure 1 illustrates the expected coverage improvement thanks to the proposed framework, i.e., a blue curve. As a baseline, we plotted the random stimulus approach with an orange curve, which is typically used in the dynamic verification domain [12]. When we start the verification, the random test may improve the coverage steadily since a number of *easy-to-activate* paths are unexplored yet. Then, if we continue to run the verification, the random test gradually decelerates since the random approach suffers from finding paths that rarely activate or scarcely affect the computational quality. On the other hand, the proposed approach incorporates the coverage-aware feedback loop into the test pattern generation. Besides, we take into account the computational quality with the integrated DUV component for the AC verification. Thanks to the CGF and DUV integration, the proposed framework is expected to accelerate the verification speed compared with the typical random testing.

The main contributions of this work include (1) the novel verification framework of AC circuits using CGF and (2) the quality-aware feedback component thanks to the DUV integration. To the best of our knowledge, this is the first work that proposes the dynamic verification framework for AC circuits with CGF[†]. Moreover, for efficiently finding test patterns that violate quality constraints, the proposed framework utilizes the DUV integration. Experimental results show that the proposed framework achieves 3.85 to 10.36 times higher coverage than the random test.

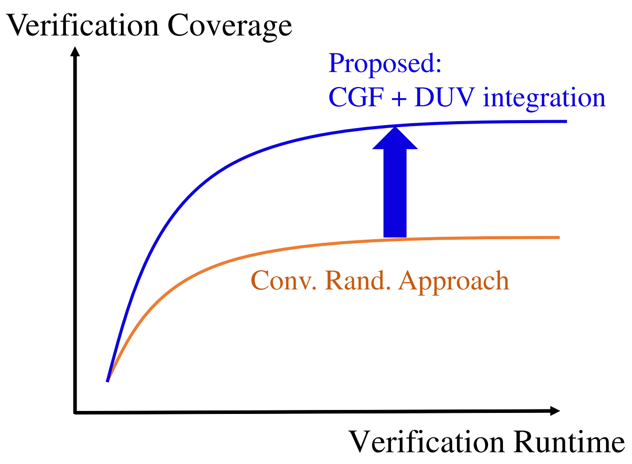


Fig. 1 Expected coverage improvement thanks to the proposed verification framework.

The rest of this paper is organized as follows. Section 2 describes the assumed verification of AC design and CGF and highlights the challenges for applying CGF to AC circuit verification. Section 3 proposes a quality verification framework of AC circuits using the CGF and DUV integration. Section 4 compares the coverage between the proposed framework and the random test. Then, this section discusses the speed-up effects of coverage improvement thanks to the proposed framework. Finally, the concluding remarks are given in Sect. 5.

2. Assumption and Challenges

This section explains the assumed verification of AC circuit and CGF and summarizes the challenges for applying CGF to the AC verification. Section 2.1 highlights the difference between the verification of the AC circuit and the conventional circuit. Then, Sect. 2.1 introduces assumed CGF. Section 2.2 discusses the challenges of CGF for verifying the AC design.

2.1 Assumed Verification of AC Design and Fuzzing

First, let us introduce the fundamental difference of the verification concept between the conventional circuit and AC circuit using Fig. 2. Remind that, we assume the dynamic verification where the computational result can be observed, the computational quality can be calculated from computational results with the given metric, and the constraint of computational quality is given and defined as the constraint which is set to the computational quality of designed AC circuit with given workloads.

In Fig. 2, three input signals InA, InB, and InC are given to the adder, and one output signal Y can be observed. Let us suppose the values of three input signals are $1001_{(2)}$, $0001_{(2)}$,

[†]A preliminary version of this work is presented in [26].

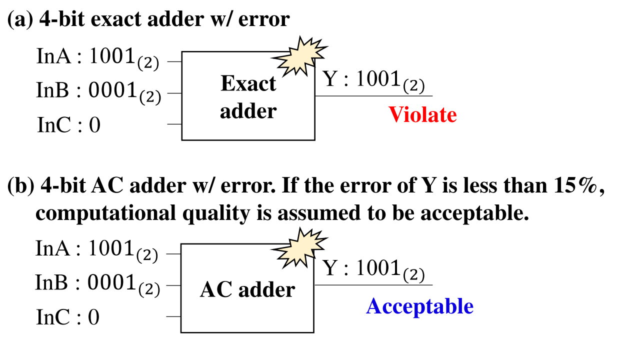


Fig. 2 (a) In the conventional circuit, any error occurrence is prohibited. (b) In the AC design, the error occurrence is allowed as long as the computational quality is acceptable.

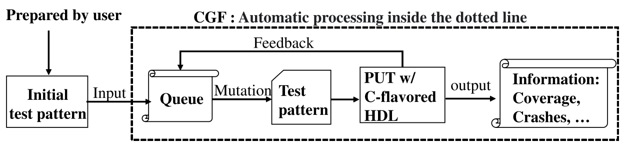


Fig. 3 Assumed CGF for C-flavored HDL. CGF automatically searches the target space by repeating (1) test pattern mutation, (2) PUT execution, and (3) feedback to the next test pattern generation.

and $0_{(2)}$. Also, in this example, the quality constraint is set to the error of Y by less than 15%. In the conventional design, any logic errors and timing errors cannot be allowed, as shown in Fig. 2(a). Hence, the verification goal is to find logic and timing errors as much as possible. On the other hand, in the AC design, the error occurrence is allowed as long as the circuit satisfies the quality constraint. For example, in Fig. 2(b), Y has the error of 10% ($=\frac{1010_{(2)}-1001_{(2)}}{1010_{(2)}}$), but this error is smaller than 15% and thus acceptable. From the above, the AC verification aims to identify errors that violate the quality constraint, which is totally different from conventional verification.

Next, this section explains the assumed CGF using Fig. 3. In CGF, the test target space is automatically searched by repeating (1) test pattern mutation, (2) PUT execution, and (3) coverage-wise feedback to the next test pattern generation [13]–[15]. As for the mutation, various strategies are developed for exploring the target space efficiently, e.g., bit-flip, byte-flip, arithmetic operation, havoc, and random operations in [13]. In the PUT execution, the activated paths can be measured via the instrumentation codes, which are inserted by the source code compile. When new paths are activated in the PUT execution, CGF adds the current input to the queue as an interesting seed for the mutation. Thanks to the lightweight mutation and coverage feedback loop, CGF improves the code coverage rapidly [16], which is quite attractive for the verification. Here, since CGF has been actively developed in the research field of software security testing [13]–[15], most of the CGF methodologies are developed for high-level languages like C language. Based on this background, this work assumes to verify the AC design starting from the C-flavored hardware description language (HDL), e.g., SystemC, CUDA C, and OpenCL.

2.2 Challenges of CGF for Verifying the AC Design

As mentioned in the previous section, CGF measures the code coverage through the PUT execution. When the coverage improves, the current input pattern is incorporated into the mutation. Therefore, the heart of CGF lies in the coverage assessment and feedback on test patterns. From this point of view, if we apply CGF to the verification of AC circuits, CGF needs to evaluate the code coverage of the AC circuit, taking into account the computation quality. However, we found that CGF originally cannot consider the computational quality by itself, which will be explained using Listing 1 as an example.

Listing 1 Code example of AC adder written in SystemC

```
1   int sc_main (int argc, char *argv[]) {
2     sc_signal<sc_uint<4>> InA, InB;
3     sc_signal<bool> InC;
4     sc_signal<sc_uint<5>> Y;
5     AC_Adder *AC_Adder1;
6     AC_Adder1 = new AC_Adder ("AC_Adder1");
7     (*AC_Adder1)(InA, InB, InC, Y);
8
9     return 0;
10  }
```

Listing 1 shows a code example of an AC adder written in SystemC, where three input signals InA, InB, and InC, and one output signal Y are used, similar to the example in Fig. 2(b). Lines 2 to 4 are the input and output declaration. Lines 5 to 7 correspond to the instantiation of the AC adder module. For simplicity, the content of AC adder module is omitted in the example. From Listing 1, we can see that the HDL of pure AC adder does not include any mechanism to evaluate how much calculation error occurs at output, e.g., Y. If CGF fails to acquire the violation condition of quality constraint, the target coverage information cannot be corrected appropriately. In this case, the mutation could not generate test patterns efficiently that violate the constraint. On the other hand, in the verification of AC circuit, we need to find test patterns that violate the computational quality constraint as discussed in Sect. 2.1. Therefore, for applying CGF to the quality verification of AC circuits, the countermeasure for evaluating the quality-aware code coverage is crucially important.

3. Proposed Verification Framework

In this section, we propose a quality verification framework for AC circuits using CGF. The key idea of the proposed framework is to incorporate the quality assessment capability into the CGF. The proposed framework embeds the DUV component into the quality calculation part of HDL and thus enables CGF to evaluate whether the quality constraint is satisfied or not. Thanks to the DUV integration, the proposed framework realizes the quality-aware feedback loop in CGF

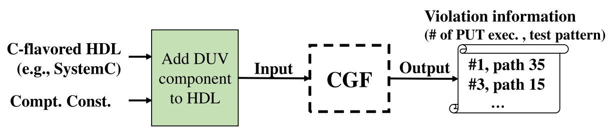


Fig. 4 An overview of the proposed framework. The proposed framework incorporates the quality assessment capability into the CGF via the Design Under Verification (DUV) integration.

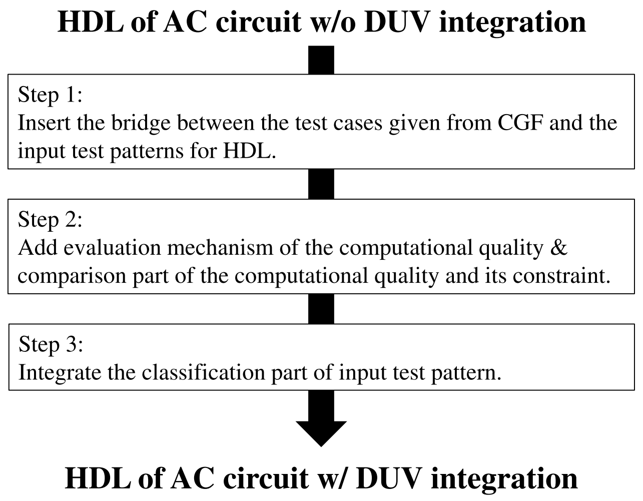
and thus quickly enhances the verification coverage for test patterns that violate the quality constraint. Note that in this paper, we focus on the test pattern-level coverage of AC circuits, whose most fine-grained definition is given as follows.

$$Cov = \frac{N^{vio}_{detect}}{N^{vio}_{total}}, \tag{1}$$

where Cov is the test pattern-level coverage, N^{vio}_{total} is the total number of test patterns that violate the constraint for the computational quality, and N^{vio}_{detect} is the number of detected test patterns that violate the constraint for the computational quality in the verification, respectively. Note that for large-scale circuits, deriving the coverage denominator (N^{vio}_{total}) may not be feasible. One possible alternative approach is to use more abstracted coverage metrics such as statement coverage, branch coverage, or conditional coverage. For example, if the verification items are defined based on the hardware design specifications and operating scenarios, we may be able to use assertion-based coverage, i.e., verification items are described using assertions, and the coverage against the assertion description is derived in the verification.

Figure 4 shows the overview of the proposed framework. As inputs, the proposed framework receives the hardware described in C-flavored HDL and the target computational quality. First, the proposed framework inserts the DUV component into the received HDL. Next, the HDL with the embedded DUV component is passed to the CGF. In the CGF, PUT is performed, and then the coverage is derived taking into account the target quality. Finally, the proposed framework outputs the results of CGF, such as the coverage and test patterns that violate the constraint.

Figure 5 shows the overview of proposed DUV integration. For the hardware description of AC circuits, the proposed framework embeds the DUV with the following three steps: insert the bridge between the test cases given from CGF and the input test patterns for HDL (step 1), add the evaluation mechanism of the computational quality, and comparison part of the computational quality and its constraint (step 2), and integrate the classification component of the test pattern based on the computational quality (step 3). The following subsection further describes the DUV integration using an example. It should be noted that the proposed DUV integration can be performed on AC circuits whose computational results can be observed and whose computational quality can be calculated from computational results according to the given quality metric.


HDL of AC circuit w/o DUV integration

Step 1:
Insert the bridge between the test cases given from CGF and the input test patterns for HDL.

Step 2:
Add evaluation mechanism of the computational quality & comparison part of the computational quality and its constraint.

Step 3:
Integrate the classification part of input test pattern.

HDL of AC circuit w/ DUV integration

Fig. 5 An overview of the proposed DUV integration. The proposed DUV integration consists of three steps.

3.1 DUV Integration for CGF

This section explains the implementation and contribution of DUV integration using an example of Listing 2, which adds the DUV component to the pure AC adder, i.e., Listing 1. Note that, in the example, the quality constraint is set to the error of Y. In Listing 2, Lines 3 to 10 correspond to the inserted bridge (step 1). Lines 15, 21 to 23, and 27 are added for evaluating the computational quality and for comparing the quality to its constraint (step 2). Lines 28 to 30 are the classification component of the test pattern (step 3).

First, we explain the bridge part between the mutated test cases and the input pattern to HDL (step 1). We highlight that this bridge is the essential part for constructing the feedback loop, as shown in Fig. 3. Let us see the example of Listing 2. First, the bridge part saves the input character string from CGF (Lines 3 and 4). Then, the stored string is converted to an integer type (Lines 5 and 6). Note that in Line 6, the integer value is normalized by 2^9. After that, in Line 10, the test pattern to the adder is updated by referring to the integer, e.g., using the pre-defined look-up table. Finally, the test pattern can be successfully given to the input of adders in Line 25. In summary, thanks to the bridge, the mutation by CGF can be reflected directly on the input test patterns to HDL.

Listing 2 Code example of AC adder with DUV integration

```
1  int sc_main (int argc, char *argv[]) {
2
3    char buf[8];
4    if (read(0, buf, 8) < 1) { exit(1);}
5    int value = *(int*) &buf[0];
6    int input = value%(1<<9);
7    int a, b, c = 0;
8
9    //Update a, b, c referring to input.
10   TestPatternGenerate(input);
11
12     sc_signal<sc_uint<4>> InA, InB;
13     sc_signal<bool> InC;
14     sc_signal<sc_uint<5>> Y;
15     sc_signal<sc_uint<5>> EX_Y;
16
17     AC_Adder *AC_Adder1;
18     AC_Adder1 = new AC_Adder ("AC_Adder1");
19     (*AC_Adder1)(InA, InB, InC, Y);
20
21     EX_Adder *EX_Adder1;
22     EX_Adder1 = new EX_Adder ("EX_Adder1");
23     (*EX_Adder1)(InA, InB, InC, EX_Y);
24
25     InA.write(a); InB.write(b); InC.write(c);
26
27     if ( |Y - EX_Y| > ERROR_TH){
28       if (input == 0){;}
29       else if (input == 1){;}
30       ...
31     }
32
33     return 0;
34   }
```

Next, the quality evaluation and test pattern classification parts are discussed (steps 2 and 3). In this example, for calculating the computational quality, the proposed framework integrates the accurate circuit component (Lines 15 and 21 to 23). Then, the computational results in the AC component are compared with the accurate result for deriving the error (Line 27). When the error exceeds the given threshold value, the input test pattern is analyzed (Lines 28 to 30). More specifically, our framework inserts conditional branches of the input test patterns for recognizing the executed pattern. For example, in Listing 2, 512 (= 2^9) branches could be enumerated at most. Since the CGF marks each branch at the compile time and traces the executed branch in PUT phase, the above branch description enables the CGF to sum up the branch coverage that violates the computational quality. From the above, thanks to the computational quality evaluation and classification of test patterns, the proposed framework realizes the quality-aware feedback loop in CGF and thus the quality verification of AC circuit.

Here, another important consideration is that the quality-aware CGF may reduce the verification time. Let us explain the expectation with Fig. 6. As previously discussed with Sect. 2.1, the AC verification aims to identify errors that violate the quality constraint. From this point of view, the total number of errors we need to find could be dramatically reduced compared with the conventional circuit. Furthermore, since the proposed DUV mechanism classifies test patterns according to the computational quality, as exemplified in lines 27 to 30 of Listing 2, CGF can perform the mutation for test patterns that violate the constraint of computational quality. Remind that CGF can utilize deterministic mutations such as bit-flip and arithmetic operation [13] that changes the test pattern locally and thus enables the neighborhood exploration. In this case, CGF can efficiently search the neighborhood of test patterns that violate the constraint. Based on the above consideration, we expect that quality-aware CGF supported by the proposed DUV accelerates the dynamic verification. Section 4.3.2 discusses the speed-up effect thanks to the quality-aware DUV integration.

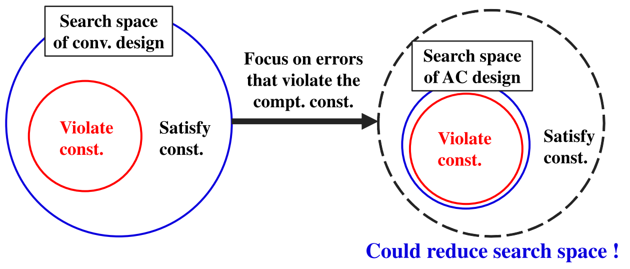

Search space of conv. design
Violate const.
Satisfy const.
Focus on errors that violate the compt. const.
Search space of AC design
Violate const.
Satisfy const.
Could reduce search space !

Fig. 6 By focusing on errors that violate the computational quality, the quality-aware CGF efficiently searches the neighborhood of test patterns that violate the constraint, reducing the verification runtime.

4. Experimental Evaluation

This section evaluates the trade-off relationship of the proposed framework between the verification runtime and coverage. Section 4.1 describes the evaluation setup. Section 4.2 shows that the proposed framework improves the verification coverage compared with conventional random testing. Then, Sect. 4.3 discusses the coverage improvement effect thanks to the proposed framework.

4.1 Evaluation Setup

In this work, we design two approximate multiply-accumulate (MAC) units, where the numbers of input bits are 3 and 4, respectively. For both MAC units, the number of accumulation clock cycles is set to 5. As the AC technique, we select the bit-width scaling, which is one of the most popular approaches for the area and power reduction [27]–[30]. Then, for the 3-bit MAC unit, we truncate the least-significant bit (LSB) of inputs. Similarly, we truncate the first and second LSBs of inputs for the 4-bit MAC unit. This circuit was implemented with the SystemC language. Next, we add the DUV component to the approximate MAC units as explained in Sect. 3.1. In this work, we add a quality constraint of the computational error in approximate MAC result. Namely, we used the following equation to derive the computational error.

$$error = \frac{|R_{ac} - R_{ex}|}{R_{ex}}, \tag{2}$$

where R_{ac} represents the computational results of approximate MAC unit, and R_{ex} is the golden results of exact MAC unit. We prepared four constraints of 0.1, 0.2, 0.3, and 0.4 for the error threshold, e.g., ERROR_TH in Listing 2. Note that the above constraints are just an example, and the proposed framework can cope with other settings in the same manner.

Then, for taking into account the quality-aware coverage, the branch insertion is performed, e.g., lines 27 to 29

Table 1 The denominator of verification coverage. We prepare 8 (= 2 × 4) different settings for the experimental evaluation. Note that the numerator of the coverage is the number of covered pairs in the verification, e.g., PUT executions.

	$error = 0.1$	$error = 0.2$	$error = 0.3$	$error = 0.4$
3 bit	11,421	11,260	8,436	4,751
4 bit	212,616	212,304	209,253	156,031

in Listing 2. Here, if we naively enumerate all the combinations of test patterns, the number of branch descriptions exponentially explodes with the number of input bits and accumulation clock cycles. For example, for the 3-bit MAC unit and the 4-bit MAC unit, when the number of accumulation clock cycles is set to 5, the maximum numbers of branch description are $2^{(3+3)\times 5} \simeq 10^9$ and $2^{(4+4)\times 5} \simeq 10^{12}$, respectively. For mitigating this issue, this paper focuses on the compressed coverage metric, which is similar to the multiplexer-aware coverage [23] or register-wise coverage [25]. More specifically, as the coverage metric, we focus on the pair of the accumulated result until the fourth clock cycle and the input test pattern in the fifth clock cycle. For example, in the 3-bit MAC unit, the multiplication result in one clock cycle ranges from 0 to 49, and the accumulated result until the fourth clock cycle ranges from 0 to 196. In this case, the maximum number of combination pairs is less than 197×64 ($=2^{(3\times 2)}$) $\simeq 10^4$, which is much smaller than $2^{(3+3)\times 5}$. It should be noted that the compressed coverage was prepared for the experimental evaluation as just an example. For large-scale circuits, this coverage metric may not be utilized since deriving the denominator of coverage may be difficult, as previously explained in Sect. 3.

Table 1 shows the numbers of target pairs, i.e., the denominator of verification coverage. For each set of the approximate MAC unit and error constraint, we preliminary evaluate whether each test pattern belonging to the pair violates the error constraint or not. If a test pattern included in the pair violates the constraint, we add the pair to the target pair, i.e., the denominator of verification coverage. Note that the numerator of the coverage is the number of covered pairs in the verification, e.g., PUT executions.

As the CGF tool, we select AFL 2.52b [13] and incorporate the AFL in the proposed design. By running the PUT in CGF, the quality-aware branch coverage can be obtained. As the initial test pattern for the CGF tool, we used one American Standard Code for Information Interchange (ASCII) character, which was randomly generated. Note that other initial test patterns and CGF tools [14], [15] can be similarly utilized in the proposed framework. As a comparison, we select a typical random approach and implement the approach using the Mersenne Twister library of the C++ language. Then, for both the proposed framework and random approach, we performed 6-hours PUT execution. Note that for each set of the approximate MAC unit and error threshold, we performed 6-hours verification. Consequently, the trade-off relationships between the runtime and quality-aware coverage are quantitatively evaluated. In this evaluation, we use a computer machine equipped with Ubuntu 16.04 LTS and AMD Ryzen Threadripper 3990X 64-Core Processor.

4.2 Evaluation Results

Figures 7 and 8 show the comparison results of the trade-off relationship of the runtime and coverage between the random test and the proposed framework. Note that Figs. 7 and 8 are the comparison results in the 3-bit and 4-bit approximate MAC units, respectively. In both figures, blue lines are the trade-off relationship of the proposed framework, and orange curves correspond to the random approach. From Figs. 7 and 8, we can see that the proposed framework significantly improves the coverage compared with the random test. For example, if we focus on the runtime of 21,600 seconds in Fig. 7(a), the proposed framework achieves the coverage of 69.79% while the coverage of random approach reaches only 18.09%. In other words, at this runtime, the proposed approach achieves 3.85 times higher coverage from 18.09% to 69.79% than the random test. Similarly, with the approximate 4-bit MAC unit, the proposed framework accelerates the verification speed and achieves 7.46 to 10.36 times higher coverage at the runtime of 21,600 seconds, as shown in Fig. 8. From these results, we experimentally confirm that the proposed framework achieves the significant coverage enhancement.

Another important observation can be seen from the comparison between Figs. 7 and 8. That is, the required runtime enlarges as the bit width increases. For example, in Figs. 7(d) and 8(d), for achieving the 20% coverage with 3-bit and 4-bit approximate MAC units, the proposed framework requires 46.28 seconds and 3,345 seconds, respectively. As summarized in Table 1, the number of target pairs significantly increases with the bit width expansion. Due to the target pair increase, the coverage improvement speed degrades. Recently, for improving the coverage or accelerating the verification speed of CGF, various techniques, e.g., hybrid fuzzing [31] and importance sampling based fuzzing [15], have been developed. Enhancing the intrinsic verification speed of the proposed framework using the above techniques is one of our future works.

4.3 Discussion

The evaluation results in Sect. 4.2 showed that the proposed framework achieved the significant coverage improvement. Let us investigate the results in detail.

4.3.1 Effectiveness of CGF

As previously discussed in Sect. 1, CGF generates test patterns for enhancing the code coverage via the mutation, which is totally different from random testing. For examining the effectiveness of the test pattern generation by CGF in more detail, we analyzed the trade-off relationship between the number of PUT iterations and the coverage.

Figures 9 and 10 show the comparison results of trade-off relationship between the number of PUT iterations and

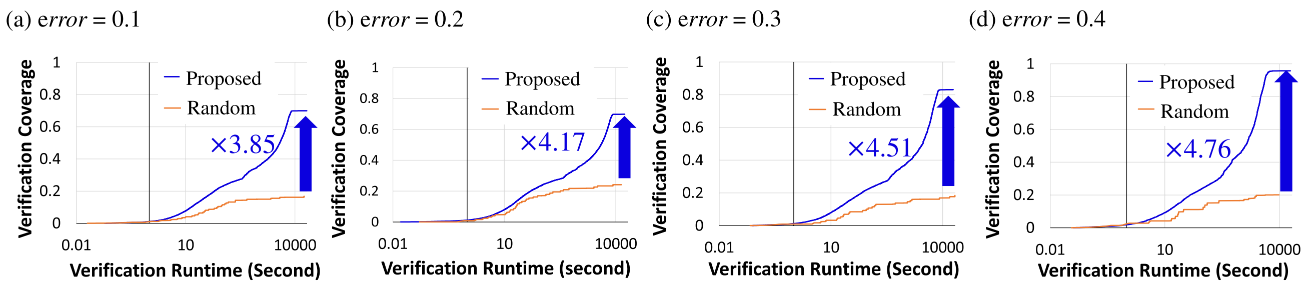


Fig. 7 Trade-off comparison between the proposed and random approach in the approximate 3-bit MAC unit. (a) $error = 0.1$, (b) $error = 0.2$, (c) $error = 0.3$, and (d) $error = 0.4$. The proposed framework significantly improves the coverage compared with the random approach.

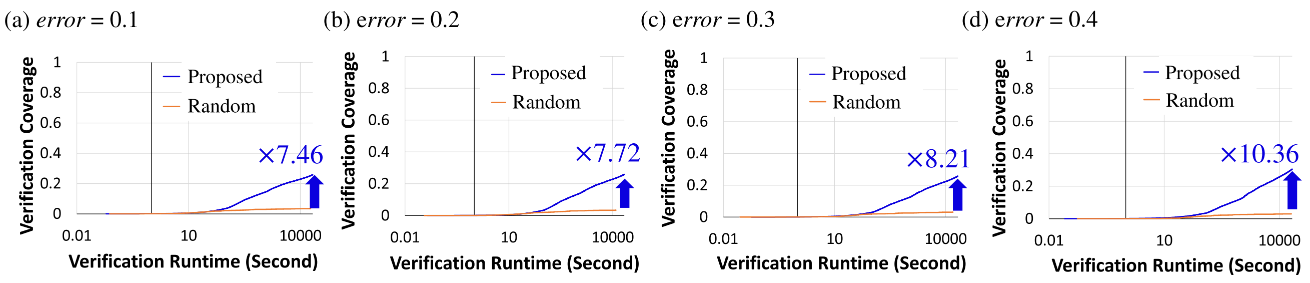


Fig. 8 Trade-off comparison between the proposed and random approach in the approximate 4-bit MAC unit. (a) $error = 0.1$, (b) $error = 0.2$, (c) $error = 0.3$, and (d) $error = 0.4$. The proposed framework significantly improves the coverage compared with the random approach.

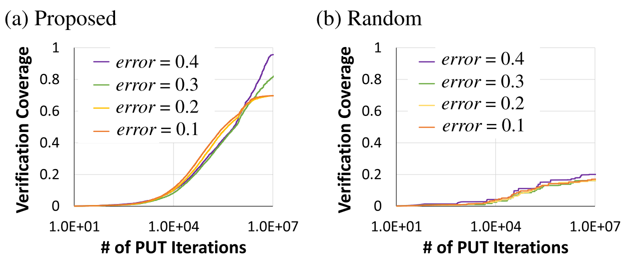


Fig. 9 Comparison of trade-off relationship between the number of PUT iterations and the coverage in the 3-bit approximate MAC unit. (a) Proposed framework and (b) Random approach. Compared to the random approach, the proposed framework improves the coverage with smaller number of PUT iteration, which indicates the effectiveness of CGF.

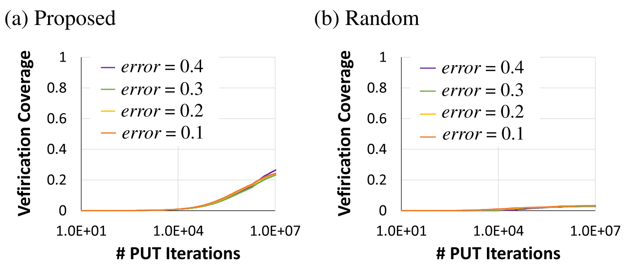


Fig. 10 Comparison of trade-off relationship between the number of PUT iterations and the coverage in the 4-bit approximate MAC unit. (a) Proposed framework and (b) Random approach. Compared to the random approach, the proposed framework improves the coverage with smaller number of PUT iteration, which indicates the effectiveness of CGF.

the coverage for 3-bit and 4-bit approximate MAC units, where Figs. 9(a) and 10(a) are the trade-off relationship of the proposed framework, and Figs. 9(b) and 10(b) correspond to the random approach. Note that we record the relationship between the number of PUT iterations and verification runtime, and thus obtain Figs. 9 and 10 from Figs. 7 and 8, respectively. From both figures, we can see that the proposed framework improves the coverage with a smaller number of PUT iterations compared to the random testing. For example, in Fig. 9(a) and (b), when the *error* constraint is set to 0.4, the proposed framework achieves the 20% coverage with 3.80 $\times 10^4$ times PUT iterations whereas the random testing requires 7.38 $\times 10^6$ times. Namely, at this coverage point, the proposed framework reduces the number of PUT iterations by second orders of magnitudes compared to the random approach. Such a significant reduction indicates that the feedback loop for test pattern generation by CGF accelerates the verification. From the above, we experimentally confirm that the proposed framework improves the coverage quickly thanks to the efficient CGF.

4.3.2 Effectiveness of the DUV Integaration

Lastly, we investigate the effectiveness of DUV integration in terms of the branch insertion strategy. As previously discussed with Fig. 6 in Sect. 3.1, the required time consumption for dynamic verification is expected to be reduced by focusing on errors that violate the quality constraint. Based on this expectation, we implement the DUV component, which does not take into account the quality constraint, e.g., remove lines 26 and 30 in Listing 2. Then, we compared the trade-off relationship between the runtime and coverage for discussing the effectiveness of quality-aware DUV

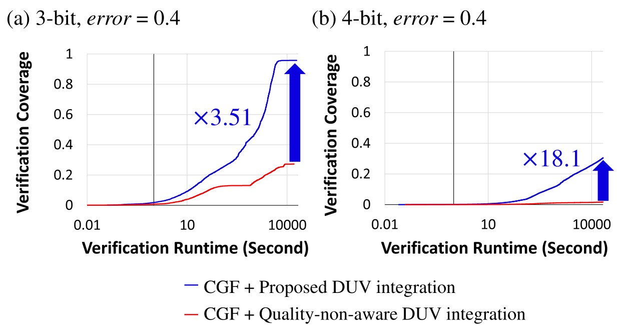


Fig. 11 Coverage improvement thanks to the quality-aware DUV integration. (a) 3-bit approximate MAC unit, and (b) 4-bit approximate MAC unit.

integration. Note that the identical CGF is utilized in the comparison. Figure 11 shows the comparison result where the *error* constraint is set to 0.4. From Fig. 11, we can see that the proposed framework achieves a better trade-off between the runtime and coverage. For example, in Fig. 11(a), the proposed framework improves the coverage from 27.23% to 95.74% by 3.51 times at the runtime of 21,600 seconds. Similarly, in Fig. 11(b), the proposed framework improves the coverage by 18.1 times at the runtime of 21,600 seconds. These results indicate that the quality-aware feedback in the CGF is well supported by the quality consideration of the proposed DUV integration. From the above, we experimentally confirm that the proposed framework improves the coverage efficiently thanks to the quality-aware CGF and DUV integration.

5. Conclusion

This paper proposed the novel dynamic verification framework of the AC circuit. The key idea of the proposed framework is to incorporate the quality assessment capability into the CGF via the DUV integration. Thanks to the integration of DUV component, the proposed framework realizes the quality-aware feedback loop in CGF and thus quickly enhances the verification coverage for test patterns that violate the quality constraint. In this work, we quantitatively compared the verification coverage of the approximate arithmetic circuits between the proposed framework and the random test. In a case study of the approximate MAC unit, we experimentally confirmed that the proposed framework achieved 3.85 to 10.36 times higher coverage than the random test.

Acknowledgments

This work was partially supported by JSPS KAKENHI Grant Number JP20K19767 and JST, PRESTO Grant Number JPMJPR20M9, Japan.

References

[1] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," Proc. ETS, pp.1–6, 2013.
[2] V.K. Chippa, S.T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," Proc. DAC, pp.1–9, 2013.
[3] Q. Xu, T. Mytkowicz, and N.S. Kim, "Approximate computing: A survey," IEEE Des. Test, vol.33, no.1, pp.8–22, 2016.
[4] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," Proc. ASPLOS, pp.301–312, 2012.
[5] R. Hegde and N.R. Shanbhag, "Soft digital signal processing," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.9, no.6, pp.813–823, 2001.
[6] A.B. Kahng, S. Kang, R. Kumar, and J. Sartori, "Slack redistribution for graceful degradation under voltage overscaling," Proc. ASP-DAC, pp.825–831, 2010.
[7] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-power digital signal processing using approximate adders," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.32, no.1, pp.124–137, 2013.
[8] S. Froehlich, D. Große, and R. Drechsler, "One method–all error-metrics: A three-stage approach for error-metric evaluation in approximate computing," Proc. DATE, pp.284–287, 2019.
[9] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler, "Precise error determination of approximated components in sequential circuits with model checking," Proc. DAC, pp.1–6, 2016.
[10] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, "MACACO: Modeling and analysis of circuits for approximate computing," Proc. ICCAD, pp.667–673, 2011.
[11] A. Bosio, S.D. Carlo, P. Girard, E. Sanchez, A. Savino, L. Sekanina, M. Traiola, Z. Vasicek, and A. Virazel, "Design, verification, test and in-field implications of approximate computing systems," Proc. ETS, pp.1–10, 2020.
[12] M. Zhou, W.N.N. Hung, X. Song, M. Gu, and J. Sun, "Temporal coverage analysis for dynamic verification," IEEE Trans. Circuits Syst. II, Exp. Briefs, vol.65, no.1, pp.66–70, 2018.
[13] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/
[14] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," Proc. ASE, pp.475–485, 2018.
[15] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," Proc. NDSS, 2017.
[16] V.J.M. Manés, H. Han, C. Han, S.K. Cha, M. Egele, E.J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," IEEE Trans. Softw. Eng., Oct. 2019.
[17] H.M. Le, D. Große, N. Bruns, and R. Drechsler, "Detection of hardware trojans in SystemC HLS designs via coverage-guided fuzzing," Proc. DATE, pp.602–605, 2019.
[18] B.P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," Commun. ACM, vol.33, no.12, pp.32–44, 1990.
[19] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence," Proc. NDSS, 2019.
[20] A. Takanen, J.D. DeMott, and C. Miller, Fuzzing for Software Security Testing and Quality Assurance, Artech House, 2008.
[21] J.E. Forrester and B.P. Miller, "An empirical study of the robustness of Windows NT applications using random testing," Proc. USEC, Aug. 2000.
[22] P. Godefroid, M.Y. Levin, and D.A. Molnar, "Automated whitebox fuzz testing," Proc. NDSS, pp.151–166, 2008.
[23] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs," Proc. ICCAD, pp.1–8, 2018.
[24] D. Ma, X. Zhang, K. Huang, Y. Jiang, W. Chang, and X. Jiao, "DEVoT: Dynamic delay modeling of functional units under voltage and temperature variations," IEEE Trans. Comput.-Aided Des. Integr.

Circuits Syst., vol.41, no.4, pp.827–839, 2022.
[25] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "DifuzzRTL: Differential fuzz testing to find CPU bugs," Proc. S&P, pp.1286–1303, 2021.
[26] K. Yoshisue, Y. Masuda, and T. Ishihara, "Dynamic verification of approximate computing circuits using coverage-based grey-box fuzzing," Proc. IOLTS, 2021.
[27] J.Y.F. Tong, D. Nagle, and R.A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.8, no.3, pp.273–286, 2000.
[28] K. Kunaparaju, S. Narasimhan, and S. Bhunia, "VaROT: Methodology for variation-tolerant DSP hardware design using post-silicon truncation o operand width," Proc. VLSID, pp.310–315, 2011.
[29] D. Kim, J. Kung, and S. Mukhopadhyay, "A power-aware digital multilayer perceptron accelerator with on-chip training based on approximate computing," IEEE Trans. Emerg. Topics Comput., vol.5, no.2, pp.164–178, 2017.
[30] I. Tsiokanos, L. Mukhanov, and G. Karakonstantis, "Low-power variation-aware cores based on dynamic data-dependent bitwidth truncation," Proc. DATE, pp.698–703, 2019.
[31] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," Proc. USENIX, 2018.

Yutaka Masuda received the B.E., M.E., and Ph.D. degrees in Information Systems Engineering from the Osaka University, Osaka, Japan, in 2014, 2016, and 2019, respectively. He is currently an Assistant Professor in Center for Embedded Computing Systems, Graduate School of Informatics, Nagoya University. His research interests include low-power circuit design. He serves on the Technical Program Committee of international conferences including ASP-DAC. He is a member of IEEE, IEICE, and IPSJ.

Yusei Honda is currently pursuing a B.E. degree in School of Informatics at Nagoya University.

Tohru Ishihara received his Dr. Eng. degree in computer science from Kyushu University in 2000. For the next three years, he was a Research Associate in the University of Tokyo. From 2003 to 2005, he was with Fujitsu Laboratories of America as a Research Staff of an Advanced CAD Technology Group. From 2005 to 2011, he was with Kyushu University and for the next seven years he was with Kyoto University as an Associate Professor. In October 2018, he joined Nagoya University where he is currently a Professor in the Department of Computing and Software Systems. His research interests include low-power design methodologies and power management techniques for embedded systems. Dr. Ishihara is a member of the IEEE, ACM and IPSJ.